So far our computer has been limited to the data that we could include right in the program. Although we have been able to control in some small way the execution of the program by means of the DIP switch settings, there has been no way to introduce data into the computer during the actual execution of the program, something that even the cheapest hand-held calculator does as a matter of course. Perhaps we should not pass off that accomplishment so lightly. We are about to develop a keyboard display program that reads the keyboard and drives the displays at the same time. This is a fairly complex program, and we think you'll gain a new appreciation for exactly what that lowly calculator goes through.

KEYBOARD DISPLAY ONE-PASS. The nature of the electronic circuits that make up the ia7301 Computer-in-a-Book make it desirable to read the keyboard with the same program that drives the displays. This is designed into the computer; it may, or may not be designed into other computer systems. That problem, however is an excellent example of reading input devices with the microcomputer. The problem is complicated because the keys tend to bounce, that is, when the key is released there is a second and possibly a third contact within a few milliseconds of releasing the key. Although these spurious contacts are very brief, on the order of a few milliseconds, their presence can foul up even the best thought out programs.

For this program, we wish the computer to scan through all of the keys once and then exit to the next instruction. Since we want the KEYBOARD DISPLAY ONE-PASS module to be set up as a subroutine, that means the normal method of exiting the subroutine will be a RET. At the same time the subroutine is scanning through the keys it is also enabling the various digits in the display. This is the nature of the circuits; it need not be written into the software. What must be built into the software is the process of loading the display circuits with the correct data out of the display buffers. If this is done properly, the very act of scanning the keyboard will also cause the displays to light with the correct data. One pass through the combined keyboard and display circuits does not make a program. That is up to the master program; without it there will not be a loop through the keyboard display subroutine and the displays will flash but will not remain lighted. When the user presses a key, the program must detect this key closure, format it into a meaningful data presentation and exit from the subroutine to a preselected address which is different from the normal return address. Thus the module will be able to exit to two different places in the CALLing program.

During the last chapter we developed a program called DELAY that passed a parameter to the subroutine simply by placing it after the CALL instruction. We intend to do the same thing now, except the passed parameter will be the address of the jump destination that program execution passes to when a key closure is detected. Because this subroutine will be of great use to us in the upcoming chapters, we will store it in the utility section of the memory. In the last chapter we stored the DELAY subroutine at location 0300H and the DISPLAY ONE-PASS subroutine at 0311H. This KEYBOARD DISPLAY ONE-PASS subroutine is so powerful that we will drop the simple display subroutine at 0311H and replace it with the subroutine that we will now develop. Load the DELAY subroutine from Chapter 7 into the computer at location 0300H. Then load KEYBOARD DISPLAY ONE-PASS at location 0311H. This subroutine will become part of our library of utility routines.

KEYBOARD DISPLAY ONE-PASS

```
0311    E3    XHTL                 ;Fetch address of jump destination
0312    23    INX    H             ;from stack.  Increment this twice
0313    23    INX    H             ;so that H/L contain adjusted return
0314    C5    PUSH   B             ;address.  Save B/C and D/E in stack.
0315    D5    PUSH   D             ;
0316    44    MOV    B, H          ;Store adjusted return address in
0317    4D    MOV    C, L          ;B/C registers.
```

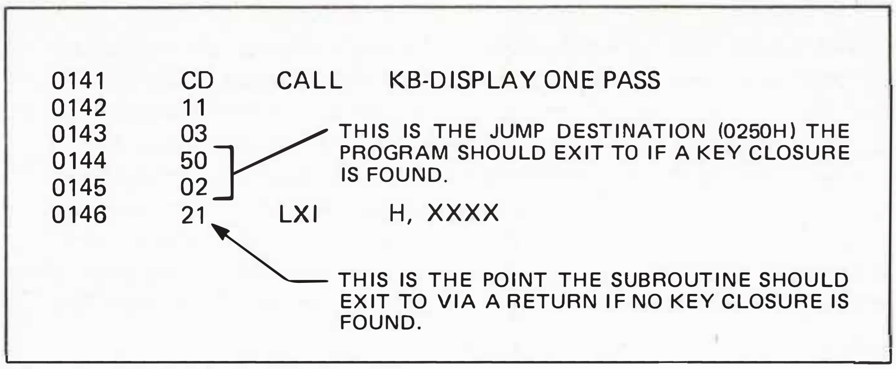
```
0318    16    MVI    D, CFH            ;Point D/E to last row of keys and
0319    CF                             ;display digit no. 7.
031A    21    LXI    H, 0007H          ;Point H/L to didplay buffer no. 7.
031B    07                             ;
031C    00                             ;
031D    7E    MOV    A, M              ;Get data from display buffer into
031E    12    STAX   D                 ;accumulator.  Then send it to
031F    CD    CALL DELAY (0300H)       ;displays.  Delay to simulate guard
0320    00                             ;circuit timing out.
0321    03                             ;
0322    00                             ;Delay constant (004F).
0323    4F                             ;
0324    1A    LDAX   D                 ;Read internal input port.
0325    EE    XRI    07                ;Complement keyboard data.
0326    07                             ;
0327    E6    ANI    07H               ;Mask out everything but keyboard data.
0328    07                             ;
0329    C2    JNZ    0343H             ;Is there a key closure?  If so, go to
032A    43                             ;0343H to see if it is real.
032B    03                             ;
032C    15    DCR    D                 ;If there is no key closure, decrement
032D    15    DCR    D                 ;D twice to point to next key row and
032E    2D    DCR    L                 ;Decrement L so H/L point to next
032F    F2    JP     031DH             ;display buffer.  If not the last
0330    1D                             ;buffer, go back to 031DH and send
0331    03                             ;next data to displays.  Loop until done.
0332    3A    LDA    001CH             ;Fetch value of bounce counter to
0333    1C                             ;accumulator and set flags.  If
0334    00                             ;counter is zero, go to 033DH to
0335    B7    ORA    A                 ;exit subroutine.  If bounce counter
0336    CA    JZ     033DH             ;is not zero, decrement it and return
0337    3D                             ;the new value to the memory location
```

```
0338    03                      ;that stores the bounce counter.
0339    3D    DCR    A          ;
033A    32    STA    001CH      ;
033B    1C                      ;Then fall through to the exit segment.
033C    00                      ;

                                ;Exit segment.
033D    69    MOV    L, C       ;This is the normal exit from the
033E    60    MOV    H, B       ;subroutine when no key closure is
033F    D1    POP    D          ;found.  Move return address back to
0340    C1    POP    B          ;H/L.  Restore B/C and D/E from stack.
0341    E3    XTHL              ;Load return address back into top
0342    C9    RET               ;of stack and restore H/L.  Return.

0343    F5    PUSH   PSW        ;Store image of key closure in stack.
0344    3A    LDA    001CH      ;Get current value of bounce counter
0345    1C                      ;into accumulator.  Set flags.  Is
0346    00                      ;it zero?  If so, this is a
0347    B7    ORA    A          ;valid key closure.  Go to 0354H to
0348    CA    JZ     0354H      ;process the key closure.  If
0349    54                      ;the key closure is not valid, fall through
034A    03                      ;to exit the subroutine.
034B    F1    POP    PSW        ;Restore the accumulator to balance the
034C    3E    MVI    A, 05H     ;stack.  Then load the accum with 05H
034D    05                      ;to initialize the bounce counter.
034E    32    STA    001CH      ;Load the new value of the bounce
034F    1C                      ;counter in the memory location
0350    00                      ;reserved for it.
0351    C3    JMP    033DH      ;Go to 033DH to exit the subroutine.
0352    3D                      ;
0353    03                      ;
```

```
0354    3E      MVI     A, 05H          ;We have reached this point because the
0355    05                              ;bounce counter was zero, and hence the
0356    32      STA     001CH           ;key closure was valid.  Store 05H,
0357    1C                              ;the initialization value in the location
0358    00                              ;reserved for the bounce counter.  The
0359    F1      POP     PSW             ;key image, saved in the stack, is
035A    E6      ANI     0EH             ;loaded into the accumulator.  Zero
035B    0E                              ;out least significant bit.
035C    07      RLC                     ;Rotate data two bit positions
035D    07      RLC                     ;to the left to open up three bits
035E    5F      MOV     E, A            ;in LSB positions.  Load this into
035F    7A      MOV     A, D            ;E.  Get row count from D and
0360    E6      ANI     ~~06H~~ 0EH     ;zero out everything but row.
0361    ~~06~~ 0E                       ;
0362    0F      RRC                     ;Shift row count into three LSB positi-
0363    B3      ORA     E               ;on.  OR with column data from E.
0364    69      MOV     L, C            ;Key image is now correct in accum.  Get
0365    60      MOV     H, B            ;address of return address from B/C.
0366    2B      DCX     H               ;Decrement to get address of jump
0367    46      MOV     B, M            ;address and load into B/C.
0368    2B      DCX     H               ;
0369    4E      MOV     C, M            ;
036A    69      MOV     L, C            ;Load the jump address into H/L.
036B    60      MOV     H, B            ;
036C    D1      POP     D               ;Restore D/E and B/C from stack.
036D    C1      POP     B               ;
036E    E3      XTHL                    ;Restore H/L from stack and put jump
036F    C9      RET                     ;address on stack.  Return.
```

This is a rather complex subroutine, but its importance dictates that we cover it in detail. Besides the problem of driving a set of displays at the same time we are reading a keyboard, we have the more general problem of writing a software module with two different exit points. One of these is used every time the displays and keyboard are scanned once but no key closure is found. This is normally done as part of a loop, either continuous or designed to terminate after a set number of scans. The module thus serves to drive the displays at the same time it is searching for a key closure. Considering the speed of the computer, most programs will place the system in one or more loops of this nature almost all of the time. When the user presses a key we want the module to exit in a different fashion to some predetermined address where the key closure can be serviced. To make the module as useful as possible, we wish to pass the jump address to the subroutine in the same way that we passed the delay constant to the DELAY subroutine of the last chapter. That means that when we CALL the keyboard display subroutine, we will follow this CALL with the address the subroutine is to exit to in the event a key closure is found.

Because of the complexity of the subroutine, it will use all of the working registers of the CPU. In order to make it more useful, we will want to save all of the registers in the stack memory so that none of the registers are out of service because of this subroutine. At the same time we have to fetch the jump address that follows the CALL in the CALLing program. This jump address will have to be saved in the event that it is needed if a key closure is found. An XTHL instruction saves the H and L registers in the stack at the same time it loads H/L with the address of the return address in the CALLing program. Actually the term return address is incorrect here, since a return at this point would be to the next memory location in the CALLing program which is the beginning of the jump destination. The actual return address is two memory locations further along in the program. See Fig. 8-1. Two INX H instructions correct the address in H/L so that they now correctly point to the desired return address in the CALLing program. The B/C and D/E registers are now saved by PUSHing them onto the stack. At this point, we have the H/L registers tied up with the return address and they are much too valuable to be used that way. Accordingly we move their contents to the B/C registers with MOV B, H and MOV C, L instructions. The registers are now saved and ready for the work of the subroutine. See Fig. 8-2.



```
0141        CD       CALL      KB-DISPLAY ONE PASS
0142        11
0143        03
0144        50
0145        02
0146        21       LXI       H, XXXX
```

THIS IS THE JUMP DESTINATION (0250H) THE PROGRAM SHOULD EXIT TO IF A KEY CLOSURE IS FOUND.

THIS IS THE POINT THE SUBROUTINE SHOULD EXIT TO VIA A RETURN IF NO KEY CLOSURE IS FOUND.

Fig 8-1. The KB-DISPLAY ONE PASS has two exit points. If no key closure is found it should return to the CALLing program two locations beyond the CALL. If a key closure is found, the program control should pass to the address immediately following the CALL, in this case, 0250H.

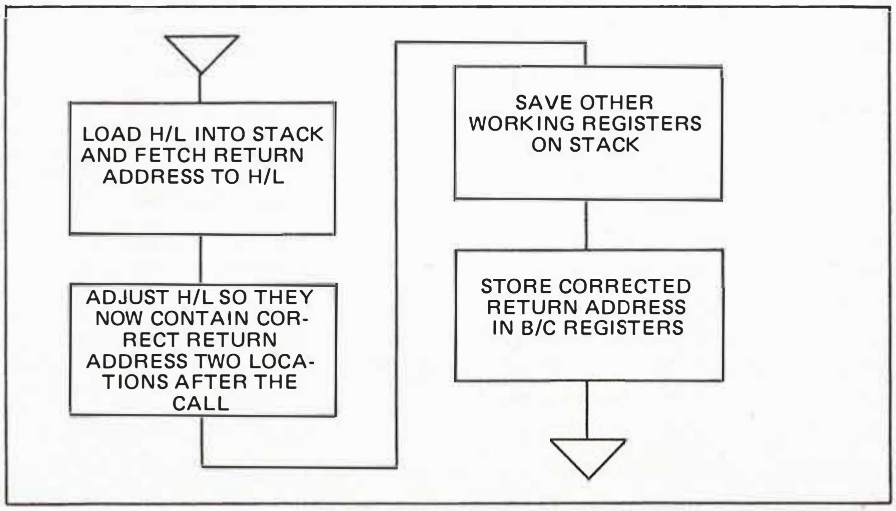

LOAD H/L INTO STACK AND FETCH RETURN ADDRESS TO H/L
ADJUST H/L SO THEY NOW CONTAIN CORRECT RETURN ADDRESS TWO LOCATIONS AFTER THE CALL
SAVE OTHER WORKING REGISTERS ON STACK
STORE CORRECTED RETURN ADDRESS IN B/C REGISTERS

Fig 8-2. The initialization segment of KB-DSPLY ONE PASS saves all of the working registers and stores the corrected return address in B/C.

The design of the ia7301 computer could have utilized a straight forward internal port structure of twenty-four input lines, each tied to a key. While this would have been simple, it would also have been expensive. To reduce costs a very common technique of setting up the keyboard as a matrix was used. This matrix of keys, eight rows high and three columns wide, is read by an internal port with eight input lines. Each of the key columns is tied to an input port line so that three of the input lines are tied to key columns; the remaining five lines are used for input signals other than the keyboard. See Fig. 8-3. Normally the keys are open and reading the internal input port will find the three columns high or at a logic one state. Even if one of the keys is depressed, the columns stay at the logic one state since they are tied electrically to a logic one

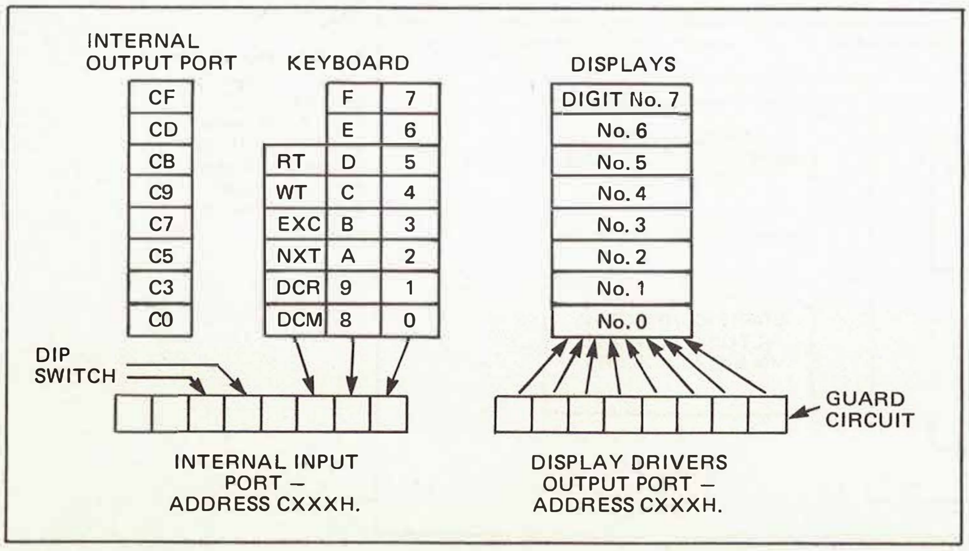


Fig 8-3 To read a key or drive the displays, a signal must be sent out from the internal output port. This activates one row of keys and one digit of the displays. The internal input port can then be read to see if there is a key closure in that row. If not, the next row is activated and checked, etc.

voltage. A second set of circuits tied to the rows of the keyboard can, row by row, pull a row of keys to the logic low or zero state. In the normal mode this is a scanning operation with one row pulled low, then the next and so on. Each time a row is pulled low, the internal port is read. Only if a key is depressed in the row that happens to be low, will the internal input port sense a logic zero in the apropriate column. The location of the key closed is found by combining the row number with the column number. The speed of the computer guarantees that many keyboard scans take place in the time it takes to press a key, so there is no chance that a key closure will occur at a time when its row has not been activated. The same circuits that drive the eight keyboard rows also drive the digits in the display. Just as only one keyboard row is activated at any given time, so only one display digit is enabled. It is up to the software module to get the correct data to the display drives so that it will be there at the same time the apropriate digit has been enabled. Although only one row of keys and one display digit are "on" at any time, the rate of scanning makes the action of the keyboards and the visual appearance of the displays such that the user senses the keyboard and displays as continuously on.

If you examine the circuit diagram in Fig. 8-3 you'll see the internal output port drive the keyboard rows and the display digits. This port is designed so that only one row is active at any given time. Furthermore, the port latches this information so that once a row is activated, it remains in that state until another row is activated. This means that one row of keys and one display digit is activated at all times. The selection of the row is determined by the address sent to the port, not by the data. Any memory write operation to one of these eight addresses will activate a row. Thus STA COXXH will activate the bottom row of keys regardless of the data in the accumulator that is sent to that address. If H/L contain COXXH, a MOV M, A instruction will accomplish the same thing.

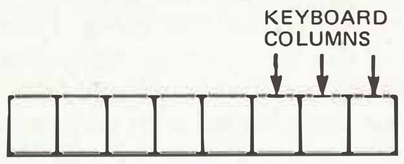
Because the output lines of the internal output port are tied to the display digits, one of these displays will always be enabled. To light up with a symbol, however, data must be sent to the display drivers. These driver circuits are connected to all of the displays so that the same data is sent to all of the digits. But, since only one of the digits is activated by the internal output port, only one of the digits will light up with the symbol. The display drivers are latching circuits so that once data is sent to them it remains in the drivers until new data is sent or until they are cleared with a special reset line. That reset line is tied to a guard circuit that automatically resets or clears the displays after a short period, on the order of a millisecond or so. The address of the display drivers ouput port is CXXXH, so that sending data to any memory location starting with C as its first address digit will load that data into the display drivers.

The fact that the internal output port is accessed by address alone and the display drivers are accessed with any address beginning with CH allows us to do both at the same time. If we send the seven segment code for F to address COH, the internal output port will activate the bottom row of keys and display digit no. 0. At the same time the display drivers will latch the data and send it to the displays. Since only digit no. 0 is activated, the numeral F will appear in that display. The others will remain blank. The numeral will continue to appear in that digit until either the guard circuit clears the display drivers or a different digit is enabled. Since it would be very confusing to have the same data appearing on one or more of the digits, in practice we make sure that the digit is enabled until the guard circuit resets the display drivers. Then and only then does the program go on to drive another digit with a different set of data. The keyboard routines in the monitor program do this by monitoring the output of the guard circuit; the keyboard and display routines that we develop for our own use simulate the action of the guard circuit by introducing a fixed delay that we are sure is longer than the guard circuit timing period. That way we know that when the delay is over, the guard circuit must have reset the displays.

The keyboard is read by an internal input port whose address is the same as the display drivers, CXXXH. If we read this address we will be able to determine if there is a key closure in the row of keys that is activated. Key closures that occur in rows that are not activated will not be picked up by the internal input port, however at the rate the computer scans down the rows of keys, it is impossible that a key closure could be missed. The fact that all three of the ports can be accessed with the same address makes this system a natural for indirect addressing. Point the D/E registers to the first row of keys with MVI D, CFH; only the two most significant digits of this address matter so we will not bother to load L. We will also need to be able to fetch data from the display buffers holding data for the displays. These were loaded with seven-segment codes by the CONVERT subroutine. Point H/L at the display buffers with LXI H, 0007H and load the data from that display buffer into the computer with MOV A, M, then send it to the displays themselves with STAX D. Since D/E contain CFXXH, the operation of sending data to the displays also enables digit no. 7 as well as the seventh or top row of keys. Before reading the keys we wish to make sure that the guard circuits have timed out so we CALL DELAY with a delay constant of 004FH as usual.

During this time the digit and the keyboard row are still enabled, even though the guard circuit has shut the display off. We can read the keyboard with an LDAX D instruction. However, the internal input port associated with this address also reads some other internal circuits so we need to format the data somewhat before operating it. During the rest of this program we are going to combine the row count being held in the D register with keyboard column input from the internal input port. The row count is expressed in terms of logic 1's, the column input in terms of logic 0's. Our first step is to convert the column input to logic 1's so row and column counts can be combined into one number. The LDAX D instruction loaded the accumulator with the data in the internal input port. This data is in the form of eight bits, the three right-hand bits being the inputs from the keyboard columns. See Fig. 8-4.

The problem of having to change selected bits of a number occurs so often, there are three instructions just for that purpose. The OR instruction can be used to set specified bits to a logic 1 without affecting the other bits. The AND instruction can be used to set specified bits to 0



BEFORE	0101 1011	HEXADECIMAL 07H = 0000 0111 IN BINARY.
AFTER XRI 07	0101 1100	XRI 07H INVERTS THE DATA IN KEYBOARD COLUMNS.
AFTER ANI 07	0000 0100	ANI 07H ZEROES OUT EVERYTHING BUT THREE RIGHT HAND BITS.

Fig 8-4. The XRI instruction complements selected bits of the accumulator. The ANI instruction zeroes out selected bits of the accumulator.

without affecting the other bits. Finally, the EXCLUSIVE OR instruction can be used to complement bits without affecting the others. The term complement means to change logical states, so that a 0 becomes a 1 and a 1 becomes a 0. This latter case is the situation at hand; we wish to complement the three right hand bits in the accumulator to make them consistent with the logic sense of the contents of the D register. The hexadecimal number 07H is the equivalent of the binary number 0000 0111. The EXCLUSIVE OR instruction at 0325H in our program performs the complementing. In every case the OR, AND and EXCLUSIVE OR instructions operate on the contents of the accumulator. XRI 07H, Exclusive OR Immediate, specifies the bits to be complimented by the immediate data that accompanies it. In this case, the three right-hand bits are specified by virtue of the fact that the binary equivalent of 07H has logic 1's in the three right hand positions. Only those bits specified by the logic 1 state will be complemented; bits specified as logic 0 will be unaffected.

Example: The accumulator contains 0011 0101. After performing an XRI 07H on it, the accumulator will contain 0011 0010.

```
Prior Contents        0011  0101
XRI 07H               0000  0111
Contents After        0011  0010
                              └── XRI 07 specifies only these bits will be complemented.
```

Since the internal input port also reads other circuits, the bits in the other five positions may contain logic 0's or logic 1's. From the point of view of the keyboard these are extraneous and need to be eliminated. The AND instruction is used to zero out specified bits. The ANI 07H, AND Immediate, at 0327H performs the ANDing function only on those bits specified by a logic 0. This is exactly the opposite of the EXCLUSIVE OR function. Since we wish to zero out the five left-hand bits of the accumulator, the immediate data accompanying the ANI instruction should be 07H, since the binary equivalent of that number has logic 0's in the five left-hand bit positions.

Example: The accumulator contains 0011 0010. After performing an ANI 07H on it, the accumulator will contain 0000 0010.

```
Prior Contents        0011  0010
ANI 07H               0000  0111
Contents After        0000  0010
                        └── ANI 07H specifies only these five bits will be zeroed out.
```

Since a key closure results in a logic 0 being loaded into the accumulator, after the XRI and ANI instructions we are left with key closures being represented in the accumulator as logic 1's, and all other non-keyboard bits as logic 0's. We can test to see if there was a key closure by performing a JNZ test Jump if Not Zero, on the contents of the accumulator. A key closure will make the computer perform the jump, in this case to 0343H, where the closure will be examined to see if it is a valid closure or only a bounce from some previous key operation.

If there is no key closure found, the JNZ test fails and the program goes on to the next row of keys and the next display digit. The internal output port activates the next row by decrementing D twice so D/E points to CDXXH. L is decremented so that it points to display buffer no. 6. This portion of the program is set up as a loop so a jump back to 031DH, the MOV A, M occurs. The basis of the looping action is a JP 031DH instruction, Jump if Positive. This is used in the same way as in the CONVERT subroutine of Chapter 6. There, you will remember, the register was decremented from 07, to 06, to 05, . . . to 00. All of these are positive numbers so the jump back through the loop occurred. Finally another decrement took the register from 00 to FF, the test failed and the program fell out of the loop. Before we discuss the method by which the program exits the subroutine after falling out of the loop, we need to look over the overall program flow. See Fig. 8-5.

The problem that we face in this module is that of ignoring key closures that are merely bounces from earlier key closures. To do that we set up a memory location 001CH, as a bounce counter. Follow the flow in Fig. 8-5 as we try a few examples. Assume the bounce counter contains 00H. The module is CALLed, the keys are scanned for a key closure (at the same time the displays are lighted), but no key closure is found. The first decision box results in a NO, so the program falls through to the box below which tests the bounce counter. Since the counter is set at 00H, this test results in the program leaving the module through a return. The bounce counter is unchanged; it still contains 00H. Since the KB-DSPLY ONE PASS module is normally used in a

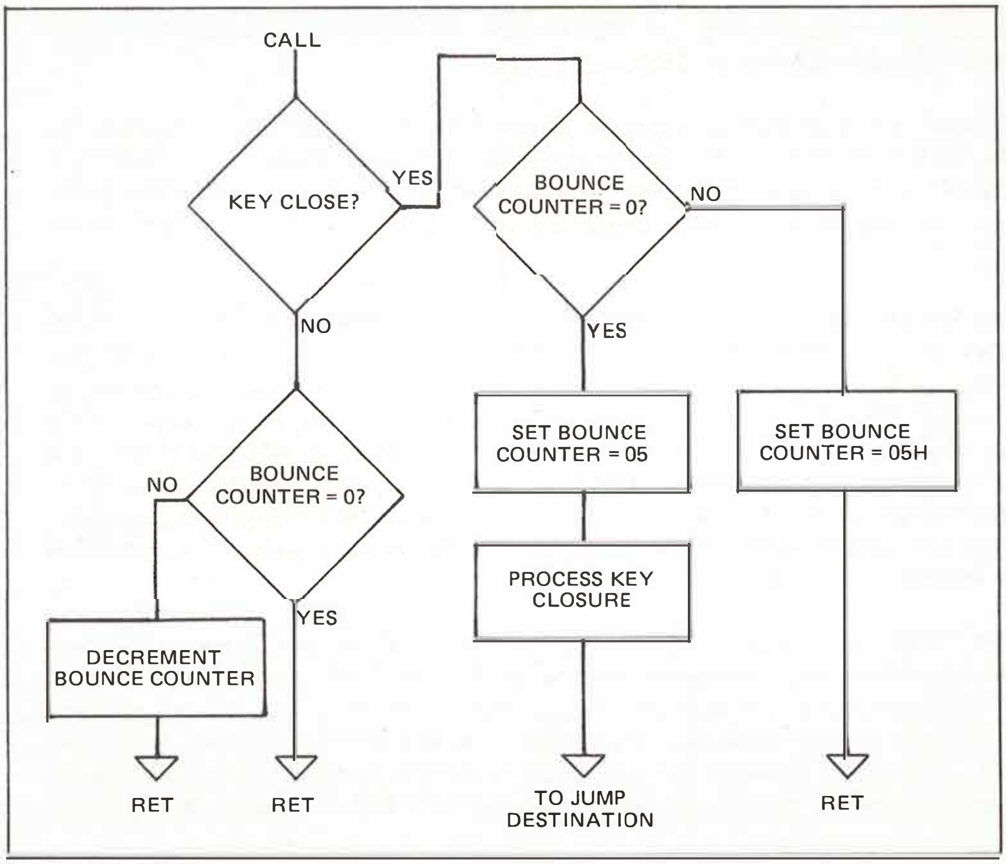


Fig 8-5. The KB-DISPLAY ONE PASS subroutine uses a bounce counter to ignore key bounces so that only valid key closures are processed.

loop, many passes through the module will occur. To the user, the displays will appear to be continuously lighted, and the bounce counter will remain at 00H.

At some point in the loop, a key closure will occur. When the first decision box is reached, the program will take the YES path and test the bounce counter with the right-hand decision box. Since the bounce counter still contains 00H, this box will result in the bounce counter being loaded with 05H and the key closure being processed and an exit to the jump destination occurring.

After the program has done whatever it is going to do with the data contained in the key closure, it will find itself sooner or later going back to the KB-DSPLY ONE PASS module. Assume that this happens so quickly the key closure from the earlier pass has not yet resulted in a bounce, so that when the Key Close?? decision box is reached, the NO path will be followed. Now, for the first time, the bounce counter is not zero. It contains 05H as a result of the exit mode from the last path when the key closure was processed. This results in the bounce counter being decremented and the normal return executed. Subsequent passes through the module will decrement the bounce counter until it reaches zero and the initial state of affairs exits again. The module is then ready to process another key closure.

But what if, after the initial key closure, a bounce had occured. This would have happened before the bounce counter could have been decremented all the way back to 00H. In that case, the system would have recognized the bounce as a key closure. Oh, the YES path would have been followed, but when the bounce counter was checked, it would have been found to contain some number, say 02H. This path requires that the bounce counter be set back to 05H and the loops continued. The key closure is seen for what it is, nothing but a bounce, and ignored.

The implications of this are that a real key closure will be processed immediately, but the system will not respond to another key closure, bounce or real, until five scans of the keyboard have produced no closures. If a closure occurs during these scans, it is ignored and another five closureless scans must take place before a key closure will be processed. Since experience has shown that five scans are sufficient to let all of the bounces die out, the module is able to respond only to real key closures. If the number of scans is increased, the action of the keyboard will become sluggish and quick, short key closures will not be recognized. Five scans seems to be the ideal compromise that eliminates bounces and yet retains the fast action keyboard.

And just how is all of this implemented in the module? We left it just as the program was falling out of the loop after searching in vain through all eight of the keyboard rows for a key closure. We now find ourselves at 0332H ready to test the bounce counter on the NO path from the Key Closure?? decision box. We load the accumulator with the bounce counter, LDA 0010H, and check it to see if it is zero or not. However, to perform that test we need to set the zero flag according to the contents of the accumulator, and just performing a data transfer like LDA will not set any of the flags. We need to do some sort of operation that will not affect the data, but will set the flags. There are a whole set of such operations, ADI 00H, SBI 00H, ANI FFH, etc., but all of these are immediate instructions and require two memory locations to store. Instead, we will use a trick that's worth remembering since you'll run into this same problem frequently.

The three logical functions we looked at earlier, OR, AND and EXCLUSIVE OR, were discussed in terms of immediate instructions, ORI, ANI and XRI. But there are counterparts of these instructions that use other registers to specify the bits that will be operated on. In the last chapter we used ORA B in the DELAY subroutine, where the contents of the B register were used to specify which of the bits in the accumulator were to be set to logic 1. We didn't present it quite that way in Chapter 7. Remember, in that subroutine we were trying to determine if the B and C registers were zero. We duplicated the contents of the C register in the accumulator and then ORed B against the accumulator. If B contained a single logic 1 in any of its eight bit positions, the ORA B would have produced a 1 in the corresponding position of the accumulator. Had any of the accumulator bits been 1 that were not covered by a 1 in the B register, they would have been unaffected and remained a 1 in the accumulator. Only if both the accumulator and B register contained all 0's prior to the ORA B; would the accumulator have contained all 0's afterwards. Thus the operation as performed in the ORA B instruction of the last chapter is totally consistent with our expanded view of the role of the OR function.

These direct data instructions, like ORA B, require only one memory location to store. In particular, ORA A ia a null instruction that will set the flags without affecting the data. This instruction is interpreted by the computer as set to a logic 1 all those bits in the accumulator specified by those bits in the accumulator that are already logic 1. Nonsense? Yes, since it will produce no change; those bits that are now 1 will remain 1 and those that are now 0 will remain 0. But it sets the flags and only requires one memory location to store it in the program. Now a JZ 033DH, Jump if Zero, instruction tests the bounce counter to see if it contains zero. If so, the program procedes to the exit portion of the subroutine. If not, a DCR A decrements the counter by one and a STA 001CH loads it back into the memory location reserved for it. This brings us to 0033DH which is the exit point for both of these branches.

In order to exit from the subroutine, all of the registers must be restored. The return address that has been saved in B and C is loaded back into H and L so that a balancing XTHL will restore both the H/L pair and load the return address back into the stack. MOV L, C and MOV H, B accomplish the B/C to H/L transfer, POP D and POP B restore the B/C/D/E registers from the stack, and XTHL restores H/L and sets up the return address. Exit via RET. All of the registers except the accumulator are exactly as they were when the subroutine was CALLed.

But what if back in the keyboard scanning loop we had found a key closure? Then the JNZ test at 0329H would have taken us to 0343H to test to see if the closure was from a bounce or was real. This is the equivalent of exiting the first decision box, Key Close??, on the YES path. Once again we have to check the status of the bounce counter, but since that test will require the use of the accumulator, we need to first save the contents of the accumulator, the key image, in the stack with a PUSH PSW instruction. The bounce counter can then be loaded into the accumulator with a LDA 001CH instruction, the flags set with a ORA A, and the bounce counter tested with a JZ instruction. If the bounce counter is not zero, we must ignore the key closure. But we had already PUSHed the key image onto the stack with a PUSH PSW instruction. Now, we have to balance the stack with an offsetting POP PSW or the rest of the data presently in the stack will never get back into the right registers when we exit the subroutine. There are ways to correct the stack pointer but that would require two memory locations, and a POP PSW requires only one memory location to store the instruction. As soon as the stack is balanced, we load 05H into the accumulator with a MVI A instruction and store that number in the memory location reserved for the bounce counter. Finally we go back to 033DH to restore the registers and exit via a return to the CALLing program.

If the bounce counter check had revealed zero contents, then for the first time in this discussion we have a bona fide key closure. The JZ test that determined this fact caused a jump to 0354H to process the closure. You'll remember that just prior to that jump we had pushed the accumulator containing the image of the key closure onto the stack. That means that now the accumulator is free, and we can operate knowing that when we are ready for the keyboard data we can simply POP it off the stack. Again, if you follow the flow chart of the module, a valid key closure requires that the bounce counter be loaded with 05H. This is done by the familiar method of a MVI A, 05H and a STA 001CH. The keyboard image is then POPed off the stack.

To process that keyboard image requires that we convert some data inherent in that image so that the result is a binary number that matches the hex numeral on the key. For example, if we press the key marked 2, we want to leave the subroutine with 0000 0010 in the accumulator since that is the binary equivalent of 2. The process by which we will accomplish this can be confusing, and the situation is not helped by the fact that we have not formally covered the binary number system yet. In order to clarify the situation, we're going to make a model that is quite a bit simpler than the actual keyboard circuits of the computer, but allows us to understand the rest of our program. That keyboard only has four keys, 0, 1, 2 and 3. You'll need a table of decimal to binary codes for these four numerals as well.

Decimal	Binary
0	00
1	01
2	10
3	11

If you examine the diagram of our simplified keyboard in Fig. 8-6, you'll notice that we have reduced the number of rows to two and the number of columns to two. However, the keys are read in exactly the same fashion as the keyboard of the Computer-in-a-Book. Pressing key no. 1 results in a signal from row 1 and column 0. Likewise, pressing key no. 2 results in a signal from row 0 and column 1. If we summarize this set of conditions, it will result in the table below:

Key	Column	Row
0	0	0
1	0	1
2	1	0
3	1	1

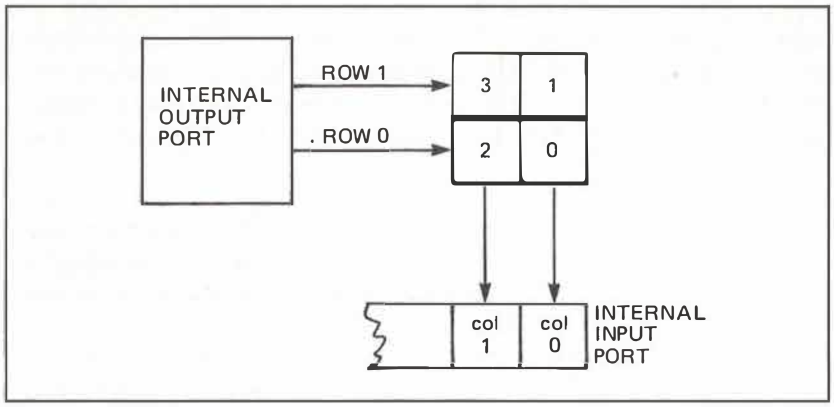


Fig. 8-6 The simplified model of the keyboard has only four keys, but operates in the same way as the larger keyboard in the Computer-in-a-Book.

Although this table exactly matches the decimal to binary table we just examined, our problem is not over yet. To say that key no. 0 puts a signal out of column 0 is true but that signal is a logic 1. To be really accurate we should extend the chart by showing both columns.

Key	Col. 1	Col. 0	Row
0	0	1	0
1	0	1	1
2	1	0	0
3	1	0	1

Now as we examine the table we find that if we take only column 1 and the row count, the conversion is correct. Key 2 results in a 1 in column 1 and a 0 in the row count, and the binary equivalent of 2 does happen to be 10. If you think we're suggesting dropping the data from column 0 completely and just using column 1 combined with the row count, you'd be right! And that is exactly what our program is going to do. We'll take the contents of the accumulator which contains the data from the two columns in the right-hand bit positions, and replace the column 1 data in the extreme right position with the row count. The result is that the accumulator then contains the binary equivalent of the four decimal switches in the keyboard. See Fig. 8-7.

And now you ask, if we were going to drop the column 0 data all along, why did we work so hard to get it into the internal input port? Because without that data we could not have detected a key closure in any of the column 0 keys, but once it has been determined that there was a key closure in the keyboard, we no longer need the column 0 data. Think of it this way. We would not have reached this point in the program if there had not been a key closure. Since there was, we know that either column 1 or column 0 has a logic 1 in them. What's more there cannot be a logic 1 in both columns; it would be physically impossible to get both keys down at exactly the same microsecond. So we know that either column 1 or column 0 has a logic 1 in it, and knowing that we no longer need the data in column 0. After all, if column 1 has a 1, then we know without looking that column 0 has a 0. And if column 1 has a 0, then column 0 must have a 1. In other words, in this system the information contained in column 0 is also contained in column 1, so all we need to know is the state of column 1.

Back to the program. When we POPed PSW off the stack we came back to the point where the keyboard image was contained in the accumulator. This data is contained in the three right-hand bit positions and represents the signals from the three keyboard columns. In the same way that we dropped the column 0 data from our model, we will drop the column 0 data from the acumulator in our real program. This is done by zeroing out the right-hand bit position with an ANI 06H. This leaves the right-hand bit position vacant and the data from columns 1 and 2 in the two

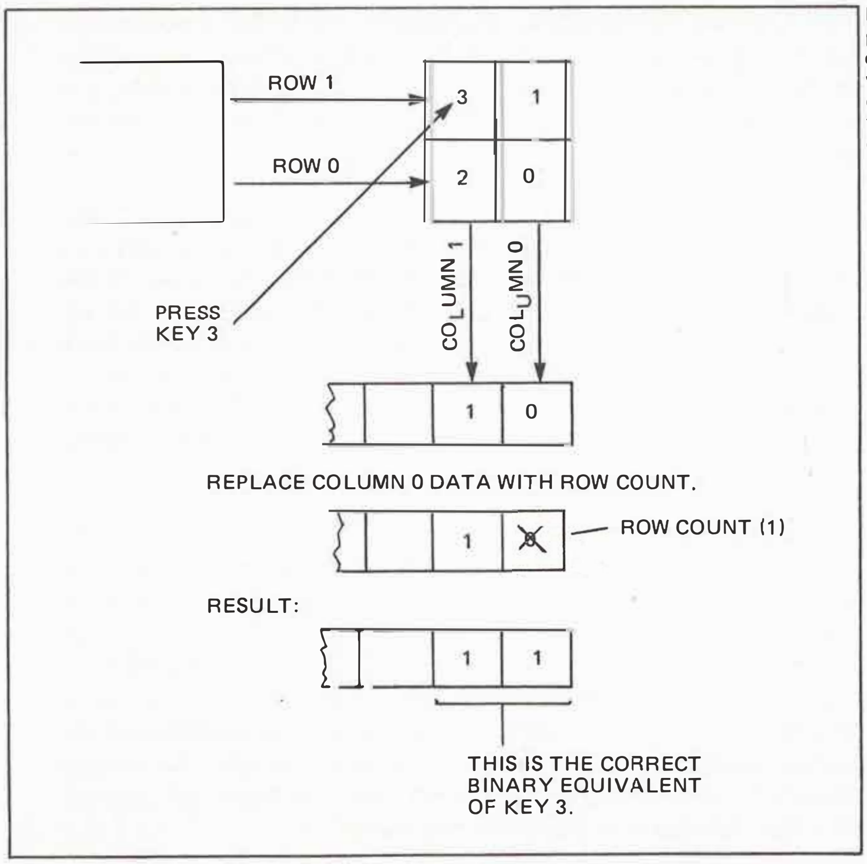

ROW 1
ROW 0
3
1
2
0
PRESS KEY 3
COLUMN 1
COLUMN 0
1
0
REPLACE COLUMN 0 DATA WITH ROW COUNT.
1
ROW COUNT (1)
RESULT:
1
1
THIS IS THE CORRECT BINARY EQUIVALENT OF KEY 3.

Fig. 8-7 By replacing the column 0 data with the row count to generate the correct binary equivalent of the key closure.

adjacent bit positions. In our model we then replaced the column 0 bit with the single bit that represented our row count. In the real computer, the row count is three bits long, so we are two bit positions short of having the room to drop that data into the right-hand vacant positions. See Fig. 8-8. Fortunately the instruction set of the computer allows us to move the bits around inside of the accumulator very easily.

The RLC instruction, Rotate Left, shifts all of the bits one bit position to the left. The bit position at the extreme left is brought all the way around and entered into the now open position at the extreme right. In this respect the accumulator becomes an endless loop connected at the ends by an invisible link. RLC moves the bits around this loop, one position per RLC instruction. The bit at the extreme left is also copied into the carry flag. The previous state of the carry flag is lost. After executing two RLC instructions, there will be three open bit positions at the right end of the accumulator, one already there and two newly created by the RLC instructions. This completes the preliminary formatting of the column data from the keyboard. Now to merge the row count from the internal output port.

This will require a slight rearranging of the row count data, and that will require the use of the accumulator. Store the columnar data in the E register with a MOV E, A and then load the accumulator with the row count which is still in the D register. This is accomplished by MOV A, D. With the row count in the accumulator we zero out all the other bits with ANI OEH which leaves only the key row count. Unfortunately, the row count is in the 1st, 2nd and 3rd bit position; the 0th bit position is vacant. Since the merger requires that the row count be dropped into the three right-most bit positions, we will have to shift the row count one position to the right with a RRC instruction, Rotate Accumulator Right. This instruction is exactly the same as RLC except the direction of rotation is to the right and not the left, and it is the extreme right-hand bit that is copied into the carry flag. We are now ready for the merger.

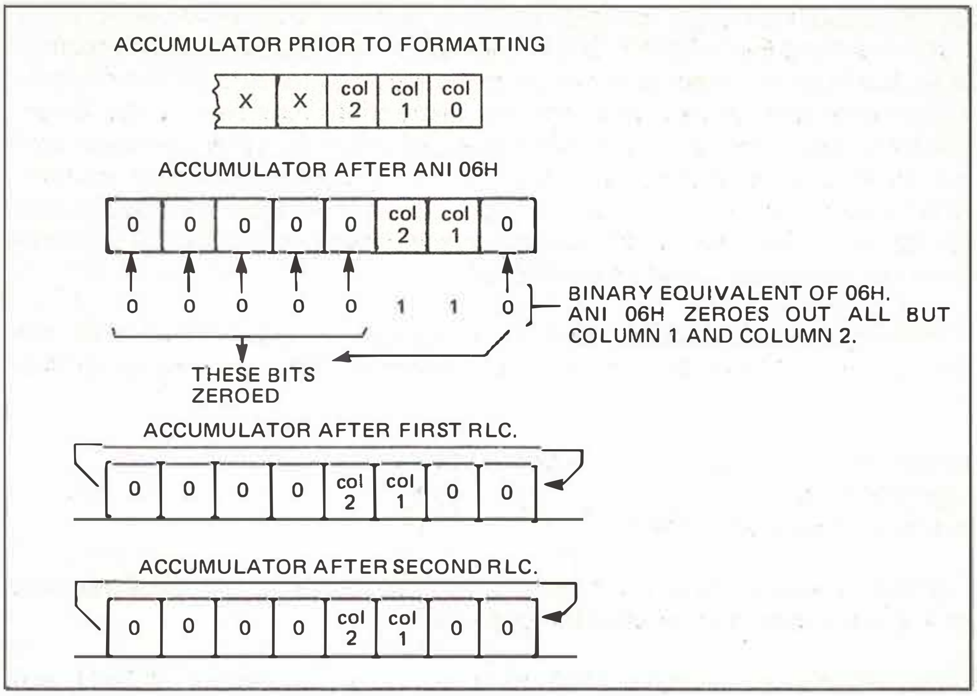

ACCUMULATOR PRIOR TO FORMATTING
X X col 2 col 1 col 0
ACCUMULATOR AFTER ANI 06H
0 0 0 0 0 col 2 col 1 0
0 0 0 0 0 1 1 0
BINARY EQUIVALENT OF 06H. ANI 06H ZEROES OUT ALL BUT COLUMN 1 AND COLUMN 2.
THESE BITS ZEROED
ACCUMULATOR AFTER FIRST RLC.
0 0 0 0 col 2 col 1 0 0
ACCUMULATOR AFTER SECOND RLC.
0 0 0 0 col 2 col 1 0 0

Fig. 8-8 Preliminary formatting of the keyboard data is done on the data from the three columns.

The formatted column data is in the E register and the formatted row data is in the accumulator. Both have been shifted in the proper directions so that there are no bit positions holding data that are common to both registers. The merger is accomplished by an ORA E instruction, OR Accumulator with E. We have already discussed the OR function and explained it as a process of selectively setting specified bits to the logic 1 state. In the process of ORA E, the bits to be set to 1 in the accumulator are specified by the contents of the E register. Remember, all three of the logical functions, OR, AND and EXCLUSIVE OR, all operate on the contents of the accumulator. Some way must be found to specify exactly which of the accumulator bits are to be operated on. Most of our experience with these logical function instructions have been of the Immediate type, such as ANI 07H, where the immediate data specified which of the accumulator bits were to be operated on. In the case of ORA E, the contents of the E register specify the accumulator bits. Wherever a bit position in the E register is a logic 1, the corresponding bit position in the accumulator will be set to 1. Wherever a bit position in the E register is a logic 0, the corresponding bit position in the accumulator will be unaffected.

Example: The accumulator contains 0000 1111, the binary equivalent of 0FH, and the E register contains 1111 0000, the binary equivalent of F0H. Perform an ORA E instruction.

Prior Contents of Accum.	0000 1111
Contents of E Reg.	1111 0000
Contents of Accum after ORA E	1111 1111

The effect is that wherever there is a 1 in either the accumulator OR the E register, there will be a 1 in the accumulator after the operation.

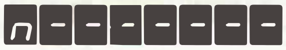
Example: The accumulator contains 1100 1010, the binary equivalent of CAH, and the E register contains 1010 1011, the binary equivalent of ABH. Perform an ORA E instruction.

Prior Contents of Accum.	1100	1010
Contents of E Reg.	1010	1011
Contents of Accum after ORA E	1110	1011

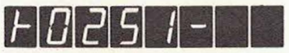
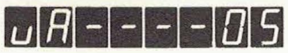
After the execution of the ORA E instruction, the contents of the accumulator is the binary equivalent of the hex key pressed by the user. We are now ready to exit the subroutine to the jump destination specified by the CALLing program. That address was modified at the beginning of the module to correct it to the return address on the assumption that a key closure was not found. We now need to readjust it so that it once again points to the jump destination in the CALLing program. MOV L, C and MOV H, B load the address back into the H/L registers and DCX H points the registers to the memory location holding the most significant hex digits of the jump destination. This is fetched to the B register with a MOV B, M. The H/L pair is pointed to the location holding the least significant hex digits with a DCX H and those digits loaded into C with a MOV C, M. The correct jump destination is now contained in the B/C registers. It is awkward to transfer control to an address in those registers so we then load the jump destination address into the H/L register pair with a couple of MOV instructions. POP D and POP B restore these registers from the stack. XTHL restores the H/L register pair from the stack in the form of a return address. When the final RET is executed, program control transfers to the jump destination and the module is complete.

Testing the Keyboard. Let's see if this elaborate routine really works. Load the bounce counter location at 001CH with 05H using the DCM mode. Then load location 0250H with a stopper, FFH. This will be our jump destination; we don't want the program running away from us when it finds a key closure. Finally, load this little test program into the system.

```
0100    CD    CALL KEYS (0311H)    ;Call KB-DSPLY ONE PASS.
0101    11                         ;
0102    03                         ;
0103    50                         ;Jump destination (0250H), when a key
0104    02                         ;closure is found.
0105    C3    JMP 0100H            ;Set up loop.
0106    00                         ;
0107    01
```

Enter:	See Displayed:
CLR EXC	

Until a key closure is found, the looping through the KB-DSPLY ONE PASS will light the displays with whatever was in the display registers. In this case, the display codes for the DCM mode are still in those registers.

5	
DCR	

The instant the 5 key is pressed, the computer recognizes it and executes the jump to 0250H where it finds an FFH which in turn stops it cold in the STEP mode. If the module operates correctly, the exit from the module should occur with the correct image of the key in the accumulator. In the case that we just tried, pressing the 5 key resulted in the accumulator containing 05H which is correct. Experiment with other keys to prove that the subroutine works consistently.

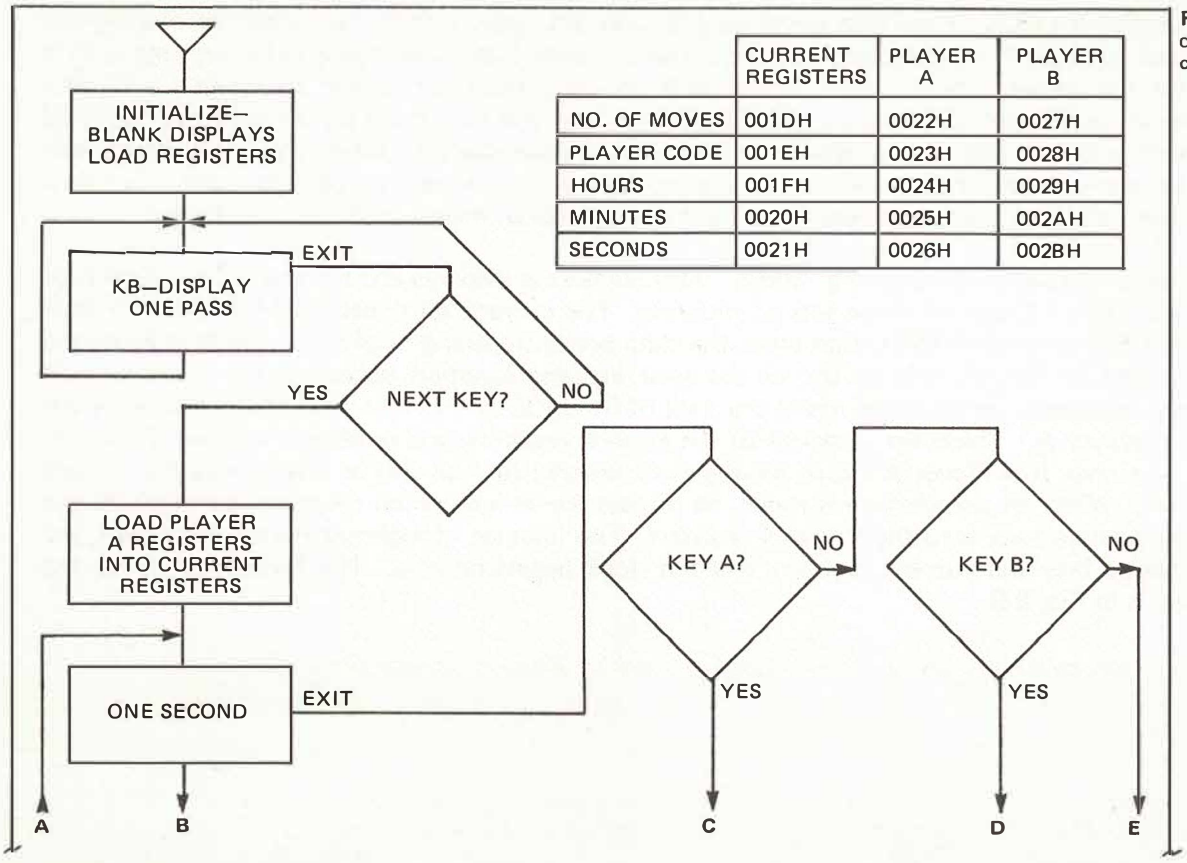
There is one other consideration when dealing with the module. When we just tested it, we first loaded the bounce counter. What would have happened if the bounce counter had happened to contain 00H prior to the CALL of the module. Try it. Load 00H into the bounce counter at 001CH, and then execute the program again. You should find that pressing the EXC key to execute the program causes the module to exit as though it had a key closure. In fact, it does. The instant the module is entered, the EXC key is recognized as a key closure since that point in the program is reached far sooner than you can remove your finger from the key. If the bounce counter has a nonzero number in it, the computer will ignore the key closure. But if the bounce counter is zero, the computer will try to process the EXC key as a valid key closure. For this reason, it is important that you initialize the bounce counter to any nonzero number at the beginning of your program. That will prevent the process of trying to execute your program from causing an exit from the module. As a final exercise, determine what value is returned in the accumulator when various keys are pressed. We will use this data later in the course.

Key Pressed	Accumulator
0	________
4	________
E	________
9	________

B	__________
DCM	__________
DCR	__________
NEXT	__________
EXC	__________
WT	__________
RT	__________

THE CHESS CLOCK Now that we've written and dubugged a utility program for reading the keyboard, let's see if we can put it to use. The program that we're going to be working with is used for tournament chess games, where each of the players is required to make a minimum number of moves per time period, usually per hour. At this time chess clocks are constructed of two electric clocks that can be started and stopped independently. While this gives the elapsed time for each player, it does not record the number of moves each player has made. Our program keeps track of both the elapsed time and the number of moves made by the players.

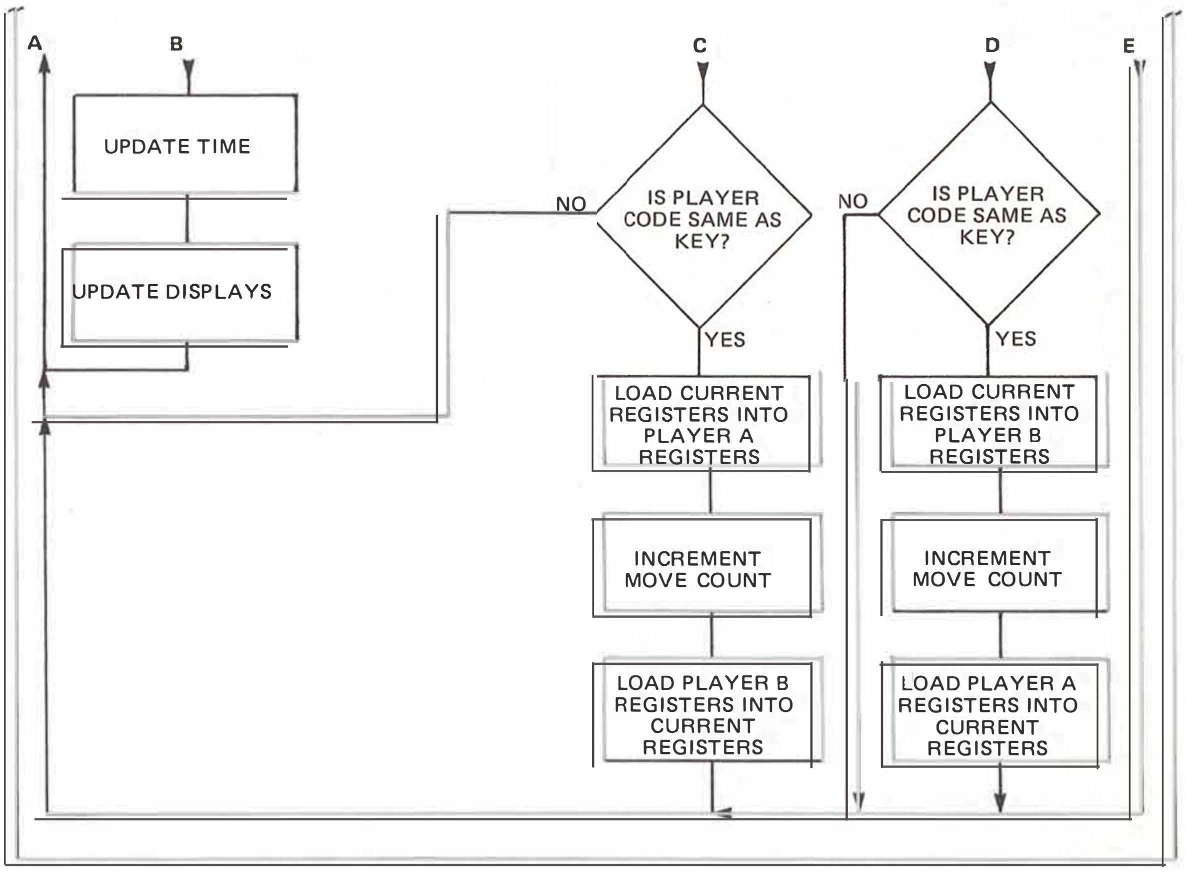
The program is begun by pressing EXC . This blanks the displays and loads all of the time registers with 00H. There are three sets of registers. The current set consists of five memory locations, 001DH through 0021H, that store the data presently being displayed. These registers are incremented as the seconds go by, so the time keeping function occurs within these current memory locations. In addition, there are a set of five locations, Player A registers, that store the data for player A. These are identical to the current registers, and during the course of the program, whenever it is Player A's turn the registers reserved for him will be loaded into the current locations. When he completes his move, he presses the A key which loads the contents of the current registers back into the Player A registers. The final set of registers, Player B registers, are then loaded into the current registers and the clock begins on him. The flow diagram for the program is in Fig. 8-9.



INITIALIZE–BLANK DISPLAYS LOAD REGISTERS
EXIT
KB–DISPLAY ONE PASS
YES
NEXT KEY?
NO
LOAD PLAYER A REGISTERS INTO CURRENT REGISTERS
ONE SECOND
EXIT
KEY A?
NO
YES
KEY B?
NO
YES
A
B
C
D
E

	CURRENT REGISTERS	PLAYER A	PLAYER B
NO. OF MOVES	001DH	0022H	0027H
PLAYER CODE	001EH	0023H	0028H
HOURS	001FH	0024H	0029H
MINUTES	0020H	0025H	002AH
SECONDS	0021H	0026H	002BH

Fig. 8-9 Flow diagram of the chess clock.


A
B
C
D
E
UPDATE TIME
UPDATE DISPLAYS
NO
IS PLAYER CODE SAME AS KEY?
YES
LOAD CURRENT REGISTERS INTO PLAYER A REGISTERS
INCREMENT MOVE COUNT
LOAD PLAYER B REGISTERS INTO CURRENT REGISTERS
NO
IS PLAYER CODE SAME AS KEY?
YES
LOAD CURRENT REGISTERS INTO PLAYER B REGISTERS
INCREMENT MOVE COUNT
LOAD PLAYER A REGISTERS INTO CURRENT REGISTERS

Load the following program into your computer.

CHESS CLOCK

```
0100    C3    JMP      0120H           ;This is the master program sequence.
0101    20                             ;The first jump is to INITIALIZE at
0102    01                             ;0120H.
0103    CD    CALL KB-DS (0311)        ;Read keyboard and loop until the
0104    11                             ;NEXT key is pressed.  That is the
0105    03                             ;signal that the game has begun.
0106    0B                             ;Jump destination for keyboard (010BH)
0107    01                             ;
0108    C3    JMP      0103H           ;This jump sets the loop.
0109    03                             ;
010A    01                             ;
010B    FE    CPI      NEXT (12)       ;A key closure has been found.  If it
010C    12                             ;is not he NEXT key go back to the
010D    C2    JNZ      0103H           ;loop.  If it is NEXT, load Player A
010E    03                             ;registers into current registers
010F    01                             ;by CALLing Load A at 037BH.
0110    CD    CALL LOAD A (037BH)      ;
0111    7B                             ;
0112    03                             ;
0113    C3    JMP DIP SW (0150H)       ;Jump to the DIP SWITCH routine to
0114    50                             ;adjust the time constant in ONE
0115    01                             ;SECOND to speed up or slow down clock.
0116    C3    CALL ONE SEC (0140H)     ;Time displays for one second.
0117    40                             ;At end of that time, update time
0118    01                             ;and displays.
0119    C3    JMP UPDATE (0200H)       ;Update time and displays.
011A    00                             ;
011B    02                             ;
```

```
011C    C3          JMP     ~~0116H~~           ;Loop back.
011D    ~~16~~ 13           0113                ;
011E    01                                      ;

0120    CD          CALL BLANK (80CCH)          ;INITIALIZE.  Blank displays.
0121    CC                                      ;
0122    80                                      ;
0123    21          LXI     H, 001CH            ;Point H/L to bounce counter.
0124    1C                                      ;Load 01H into bounce counter so
0125    00                                      ;the EXC key will not cause exit
0126    36          MVI     M, 01H              ;from KB DSPLY ONE PASS when program
0127    01                                      ;first entered.
0128    01          LXI     B, 000FH            ;
0129    0F                                      ;Load B with 00H and
012A    00                                      ;C with 0FH.  C will act as loop cntr,
012B    0D          DCR     C                   ;and B as data.  Decrement loop
012C    23          INX     H                   ;counter and increment memory pointer
012D    70          MOV     M, B                ;to next register.  Load 00H into
012E    C2          JNZ     012BH               ;memory register.  If last of registers
012F    2B                                      ;fall out of loop.  If not, go back to
0130    01                                      ;012BH.
0131    3E          MVI     A, A0H              ;Store A0H in the Player A player
0132    A0                                      ;code register at 0023H.
0133    32          STA     0023H               ;
0134    23                                      ;
0135    00                                      ;
0136    3E          MVI     A, B0H              ;Store B0H in the Player B player
0137    B0                                      ;code register at 0028H.
0138    32          STA     0028H               ;
0139    28                                      ;
013A    00                                      ;
013B    C3          JMP     0103H               ;Go back to the master program.
013C    03                                      ;
```

```
0140    06    MVI     B, A0H          ;ONE SECOND.  Store timing
0141    A0                            ;constant, A0H, in B register.
0142    CD    CALL KB-DS (0311H)      ;Make one pass through KB-DSPLY ONE
0143    11                            ;PASS.  If there is a key closure, exit
0144    03                            ;to 0275H to process it.  If no
0145    75                            ;closure, return to 0147H and decrement
0146    02                            ;loop counter.
0147    05    DCR     B               ;When loop counter is down to zero, go
0148    CA    JZ      0119H           ;back to main program at 0119H.  If
0149    19                            ;loop conter is not zero, loop back
014A    01                            ;to 0142H and repeat.
014B    C3    JMP     0142H           ;
014C    42                            ;
014D    01                            ;

0150    3A    LDA     C000H           ;DIP SWITCH.  Load DIP switches into
0151    00                            ;accum.  This address contains other
0152    C0                            ;data as well so that will have to be
0153    E6    ANI     30H             ;masked out.  After ANI 30H, accum will
0154    30                            ;contain only DIP switch data.
0155    CA    JZ      0116H           ;If neither switch is closed, go back
0156    16                            ;to master program.
0157    01                            ;
0158    FE    CPI     30H             ;Are both switches closed?  If so, ignore
0159    30                            ;them and return to master program.
015A    CA    JZ      0116H           ;
015B    16                            ;
015C    01                            ;
015D    FE    CPI     20H             ;Is the left switch closed?  If so,
015E    20                            ;go to the speed up segment.  If not,
015F    CA    JZ      0169H           ;the only other possibility is that
0160    69                            ;the other switch is closed.
0161    01                            ;
```

```
0162    21    LXI    H, 0141H     ;Slow Down segment.  Point H/L to
0163    41                        ;timing constant in ONE SECOND.
0164    01                        ;
0165    34    INR    M            ;Increment timing constant, and go
0166    C3    JMP    0116         ;back to master program.
0167    16                        ;
0168    01                        ;
0169    21    LXI    H, 0141H     ;Speed up.  Point H/L to timing constant
016A    41                        ;in ONE SECOND.
016B    01                        ;
016C    35    DCR    M            ;Decrement the timing constant and
016D    C3    JMP    0116H        ;go back to the master program.
016E    16                        ;
016F    01                        ;

0200    21    LXI    H, 0021H     ;UPDATE TIME AND DISPLAYS.  Point H/L to
                                   current seconds register.
0201    21                        ;
0202    00                        ;
0203    01    LXI    B, 6010H     ;Load B with 60H and C with 10H.
0204    10                        ;
0205    60                        ;
0206    7E    MOV    A, M         ;Fetch current seconds count from
0207    C6    ADI    01H          ;memory and increment by one.
0208    01                        ;Adjust to make it a decimal number.
0209    27    DAA                 ;and load it back into the current
020A    77    MOV    M, A         ;seconds register in the memory.
020B    B8    CMP    B            ;Does this make the seconds count 60?
020C    C2    JNZ    0226H        ;If not, go to display portion of
020D    26                        ;program.  If 60, load 00H into the
020E    02                        ;current seconds register.
020F    74    MOV    M, H         ;
```

```
0210     2B     DCX    H              ;Point H/L to current minutes register.
0211     7E     MOV    A, M           ;Fetch current minutes count from
0212     C6     ADI    01H            ;memory and increment by one.
0213     01                           ;Adjust to make it a decimal number
0214     27     DAA                   ;and load it back into the current
0215     77     MOV    M, A           ;minutes register in the memory.
0216     B8     CMP    B              ;Does this make the minutes count 60?
0217     C2     JNZ    0226H          ;If not, go to display portion of
0218     26                           ;program.  If 60, load 00H into the
0219     02                           ;current minutes register.
021A     74     MOV    M, H           ;
021B     2B     DCX    H              ;Point H/L to current hours register.
021C     7E     MOV    A, M           ;Fetch current hours count from
021D     C6     ADI    01H            ;memory and increment by one.
021E     01                           ;Adjust to make it a decimal number
021F     27     DAA                   ;and load it back into the current
0220     77     MOV    M, A           ;hours register in the memory.
0221     B9     CMP    C              ;Does this make the hours count 10?
0222     C2     JNZ    0226H          ;If not, go to display portion of
0223     26                           ;program.  If 10, load 00H into the
0223     02                           ;current hours register.
0225     74     MOV    M, H           ;

0226     11     LXI    D 001DH        ;DISPLAY. Point D/E to current
0227     1D                           ;number of moves register.
0228     00                           ;
0229     21     LXI    H, 000FH       ;Point H/L to left-most display register.
022A     0F                           ;
022B     00                           ;
022C     0E     MVI    C, 02H         ;Load C with 02H to set it up for SUB3.
022D     02                           ;
022E     1A     LDAX   D              ;Load accum with current number of moves.
```

```
022F    CD    CALL SUB3 (80A5H)     ;SUB3 loads contents of number of
0230    A5                          ;moves into two left-most displays.
0231    80                          ;
0232    13    INX     D             ;Point D/E to current player code
0233    2B    DCX     H             ;register.  Point H/L to display
0234    2B    DCX     H             ;register no. 5.
0235    1A    LDAX    D             ;Load accum with current player code.
0236    47    MOV     B, A          ;Save player code in B.
0237    13    INX     D             ;Point D/E to current hours register.
0238    1A    LDAX    D             ;Load accum with current hours.
0239    B0    ORA     B             ;Combine player code, left digit, and
023A    CD    CALL SUB3 (80A5H)     ;hours, right digit in accum.  SUB3
023B    A5                          ;loads these into display registers
023C    80                          ;no. 5 and no. 4.
023D    13    INX     D             ;Point D/E to current minutes register.
023E    2B    DCX     H             ;Point H/L to display register no. 3.
023F    2B    DCX     H             ;
0240    1A    LDAX    D             ;Load accum with current minutes.
0241    CD    CALL SUB3 (80A5H)     ;Load current minutes into display
0242    A5                          ;registers no. 3 and no. 2
0243    80                          ;
0244    13    INX     D             ;Point D/E to current seconds register.
0245    2B    DCX     H             ;Point H/L to display register no. 1.
0246    2B    DCX     H             ;
0247    1A    LDAX    D             ;Load display registers no. 1 and no. 0.
0248    CD    CALL SUB3 (80A5H)     ;with current seconds count.
0249    A5                          ;
024A    80                          ;
024B    CD    CALL CONVERT (8132H);Convert contents of display registers
024C    32                          ;into seven segment code in the display
024D    81                          ;buffers.
024E    C3    JMP 011CH             ;Go back to master program.
024F    1C                          ;
0250    01                          ;
```

```
0275    FE    CPI     0AH              ;KEYBOARD SERVICE.  Come here when a
0276    0A                             ;key closure is found.  If not A key,
0277    C2    JNZ     028BH            ;go to 028BH to see if it is B key.
0278    8B                             ;If it is the A key, load accum with
0279    02                             ;the current player code and check to
027A    3A    LDA     001EH            ;see if it is the code for player A.
027B    1E                             ;If it is not the code for player A,
027C    00                             ;the user has pressed the wrong key
027D    FE    CPI     A0H              ;and the program will ignore it and
027E    A0                             ;go back to the master program at 0116H.
027F    C2    JNZ     0116H            ;If both the code and the key are A,
0280    16                             ;then player A must have signaled the
0281    01                             ;computer that he has just moved.
0282    CD    CALL RESTA (0396H)       ;Load the current registers into the
0283    96                             ;Player A registers.
0284    03                             ;
0285    CD    CALL LOADB (0375H)       ;Load the current registers with the
0286    75                             ;contents of the player B registers.
0287    03                             ;
0288    C3    JMP     0116H            ;Go back to the master program.
0289    16                             ;
028A    01                             ;
028B    FE    CPI     0BH              ;The key closure was not A.  Is it B?
028C    0B                             ;If not, the user has pressed some
028D    C2    JNZ     0116H            ;key other than A or B and the computer
028E    16                             ;should ignore it.  Go back to the
028F    01                             ;master program.
0290    3A    LDA     001EH            ;If B was pressed, check to be sure
0291    1E                             ;the current player code is also B.
0292    00                             ;If it is not B, the user has made a
0293    FE    CPI     B0H              ;mistake and the computer should ignore
0294    B0                             ;it and go back to the master pro-
```

```
0295    C2    JNZ    0116H          ;gram.  If both the code and the key
0296    16                          ;are B, then the player B must have signaled
0297    01                          ;the computer that he has just moved.
0298    CD    CALL RESTB (0390H)    ;Load the current registers into the
0299    90                          ;Player B registers.
029A    03                          ;
029B    CD    CALL LOADA (037BH)    ;Load the current registers with the
029C    7B                          ;contents of the Player A registers.
029D    03                          ;
029E    C3    JMP    0116H          ;Go back to master program.
029F    16                          ;
02A0    01                          ;

0375    01    LXI    B, 0027H       ;LOADB.  Point B/C to first of Player B
0376    27                          ;registers.
0377    00                          ;
0378    C3    JMP    037EH          ;Skip over the initialization for
0379    7E                          ;player A to 037EH.
037A    03                          ;
037B    01    LXI    B, 0022H       ;LOADA.  Point B/C to first of Player A
037C    22                          ;registers.
037D    00                          ;
037E    21    LXI    H, 001DH       ;Point H/L to first of current registers.
037F    1D                          ;
0380    00                          ;
0381    16    MVI    D, 05H         ;Set up D as a loop counter.
0382    05                          ;
0383    0A    LDAX   B              ;Load accum with contents of Player
0384    77    MOV    M, A           ;register.  Then transfer to current
0385    15    DCR    D              ;register.  Decrement loop counter.
0386    23    INX    H              ;Point H/L at next current register.
0387    03    INX    B              ;Point B/C at next player register.
```

```
0388    C2    JNZ     0383H       ;If not last register, go back
0389    83                        ;through loop again.
038A    03                        ;
038B    C9    RET                 ;Return to master program.

0390    01    LXI     B, 0027H    ;RESTB.  Point B/C to first of player
0391    27                        ;B registers.  Then jump on over the
0392    00                        ;initialization portion of RESTA to
0393    C3    JMP     0399H       ;0399H.
0394    99                        ;
0395    03                        ;
0396    01    LXI     B, 0022H    ;RESTA.  Point B/C to first of player
0397    22                        ;A registers.
0398    00                        ;
0399    21    LXI     H, 001DH    ;Point H/L to first of current registers.
039A    1D                        ;
039B    00                        ;
039C    7E    MOV     A, M        ;Load current number of moves register into
039D    C6    ADI     01H         ;the accum and increment it by one.
039E    01                        ;
039F    27    DAA                 ;Adjust to decimal number.
03A0    02    STAX    B           ;Store in player register for number
03A1    03    INX     B           ;of moves.  Point B/C to next player
03A2    23    INX     H           ;register.  Point H/L to next current
03A3    16    MVI     D, 04H      ;register.  Set D register up as loop
03A4    04                        ;counter.
03A5    7E    MOV     A, M        ;Load current register into the
03A6    02    STAX    B           ;accum and then transfer to the player
03A7    15    DCR     D           ;register.  Decrement loop counter.
03A8    23    INX     H           ;Point H/L to next current register.
03A9    03    INX     B           ;Point B/C to next player register.
```

```
03AA      C2      JNZ      03A5H          ;If not last register, take another
03AB      A5                              ;pass through the loop.  If done,
03AC      03                              ;return to the master program.
03AD      C9      RET                     ;
```

In addition, be sure that the DELAY and the KYBD-DSPLY ONE PASS subroutine already discussed are loaded into locations 0300H and 0311H.

The first jump out of the master program is to the initialization program segment at 0120H. Here the display registers are loaded with the code to produce blanks in the displays. Then the bounce counter memory location at 001CH is loaded with an 01H so that the EXC key will not cause an exit from the keyboard subroutine. Otherwise, If this memory location happened to contain 00H at the beginning of the program, pressing EXC to execute our program would cause an exit from the keyboard routine and foul up the entire program. Then 00H is loaded into the next 15 registers by setting up C as a loop counter initialized at 0FH (15 in decimal) and B as the data register by storing 00H there. Then a MOV M, B loads 00H in each of the memory locations, while the contents of C serve as a counter to be sure only the desired memory locations are zeroed out. Two of the memory locations serve to hold the player identifying code, so that when the contents of the registers are stored in the current registers being displayed, the user can see which player is represented on the displays. These are loaded with the apropriate codes, A0H for player A and B0H for player B, through a pair of STA instructions. This completes the initialization portion of the program.

The next portion of the program tests to be sure the game is begun. The players cue the computer by pressing NEXT when the game actually begins. This occurs in the master program by a CALL to the keyboard subroutine and an immediate jump back to the loop. The exit point for the keyboard subroutine is specified after the CALL as 010BH which is the next address in the master program following this little loop. As a result of these instructions the computer displays the contents of the display registers continually (blanks), waiting for a key closure. When that closure comes, a jump to 010BH occurs immediately. There the key image, now stored in the accumulator, is tested to see if it is NEXT. From the exercise you did earlier on the keyboard, you know that the NEXT key exits from the keyboard subroutine with a 12H in the accumulator. We test for NEXT with a CPI 12H and if the test fails, one of the players must have pressed a key in error so we ignore it by jumping back into the loop. If the test is successful, the game has begun. The computer automatically assigns the first player to move the lable of Player A. It loads the contents of the Player A registers into the current registers so they can be displayed and the Player A clock begun.

The Player A registers are loaded into the current registers with a subroutine labeled LOADA located at 037BH. That subroutine points the B/C pair to the first of the player A registers at 0022H and the H/L pair to the first of the current registers at 001DH. Register D is set up as a loop counter by initializing it at 05H since there are five registers to be transfered. An LDAX B instruction loads the accumulator with the contents of the memory location pointed to by the B/C pair. This instruction is identical to the LDAX D instruction already discussed. Then a MOV M, A loads the data into the apropriate current register. The loop counter is decremented by DCR D and the current register pointer is incremented with an INX H. An INX B also increments the B/C pair that are pointing to the player register. This instruction operates exactly like the INX H instruction. When all five of the transfers have taken place a RET is executed to take the program execution back to the master program.

Since we are about to perform the time keeping function on the computer we will use the DIP SWITCH subroutine worked out in the last chapter to adjust the timing constant to make the clock accurate. This is done by a jump from the master program. Since the routine is identical to the one discussed earlier we will pass on to the next point. That is the ONE SECOND subroutine at 0140H which is almost identical to the version worked out in the last chapter. When that subroutine is done, it returns control to the master program which jumps to the program segment at 0200H.

This portion of the program is designed to do the incrementing of those registers assigned to displaying the time. It also loads the registers into the apropriate display registers and does the conversion to seven-segment codes and loads that code into the display buffers. If you have been following the course so far, this portion of the program should be very clear to you. It increments and does a DAA on each of the three time registers in the current register lineup. This portion of the program is very similar to the UPDATE TIME module of Chapter 7. The only difference is that the time is cut off at 10 hours instead of 24 hours. This is because there is not enough room in the displays for two digits of hours time. It is assumed that anyone playing over ten hours will be aware of the fact, and can mentally add ten hours to the count that appears on the displays.

After the three current registers that contain the current time count have been incremented and adjusted, it is time to display them. This is done in that program segment located at 0226H. This too is similar to a module of Chapter 7. There is one twist, however, that makes it worth while to go through the program in a little more detail. All three of the register sets, the current registers, the Player A registers and the Player B registers, are arranged in the same order. First, at the lowest memory location is the number of moves each player has made, and may run into two decimal digits. Next is the player code. While this takes an entire register, only the most significant digit is used to store the code, either A0H for Player A or B0H for player B. The next register is the hours register which keeps track of how many hours the player has spent deliberating and executing his moves. Since we have written the program so that this register resets when the counter reaches 10, the display will never go higher than 9. As that count is incremented it returns to 0. This means that only one digit will be needed to display the hours count. The remaining two registers are used for the minutes and the seconds count, in that order. At any given time during the execution of the program, one of the sets of registers belonging to one of the players will be stored in the current registers. Only the current registers appear in the displays. When the player deliberating makes his move, he presses a key. When the computer recognizes the key closure and makes sure the player has not made an error, it loads the current registers back into the player registers, and then loads those of the other player into the current register. During the deliberation time, the values in the current registers are being incremented every second so that they keep a running count on the total each player has spent in play.

The program segment at 0226H points D/E to the first of the current registers, the number of moves register. H/L are pointed to the display register corresponding to the left-most display. Register C is loaded with 02H, so everything is primed for performing a CALL SUB3 which will load the contents of the current number of moves register into the two left-most display registers. The D/E pair is incremented to point it to the next current register, in this case the current number of moves register. H/L are then decremented twice so that they point to display register no. 5. We want to use SUB3 again, but that subroutine requires that the numbers to be displayed are contained in the accumulator. LDAX D loads the accumulator with the contents of the current player code register. We now have to somehow merge this data with the data in the current hours register. To do this we temporarily store the player code in the B register with a MOV B, A instruction. INX D points D/E to the current hours register, and a second LDAX D loads the accumulator with that data. Since the player code occupies only the most significant digit of the B register, and the current hours count occupies the least significant digit of the accumulator, we can combine the two in the accumulator with a ORA B instruction. A CALL SUB3 then loads this data into display registers no. 5 and no. 4. The remainder of the registers containing the current minutes and seconds count are loaded into the display registers in the same way as the number of moves register. All that remains is to CALL CONVERT to load the seven-segment code equivalent of the display registers into the apropriate display buffers. A jump back to the beginning of the ONE SECOND portion of the master program starts the process over. In this mode, the clock is displaying the current number of moves already made by Player A, zero at this point, the player code, and the time spent by this player so far. This will continue until player A makes a move and presses his key, A .

When a player presses a key, the implication is that he has made his move, and his clock should stop and that of the other player shoud begin. First, however the computer must be sure that the player has pressed the right key. It does this by pulling out the contents of the current player code from the current registers, and then comparing that code to the key that was just pressed. If there is not a match, a key must have been pressed in error and the computer will ignore the key closure. To do this it CALLS one of two subroutines, either RESTA or RESTB depending upon which player is in the current registers. If the current player is Player A, RESTA is CALLed at 0396H. The role of this part of the program is to load the contents of the current registers back into the player registers. In the case of RESTA, the current registers are loaded into the Player A registers. In the process, the number of moves register is incremented and adjusted to a decimal number. Since this is a very straight forward segment we won't go into it here, except to point out that RESTB uses most of the same instructions. In fact, the only difference between the two is that in one the B/C pair is pointed to 0027H, the first of the Player B registers. In order to eliminate the duplication of most of the instructions, one subroutine is used with two different entry points. One of these points is labeled RESTA, the other RESTB. The only difference between the two , is that RESTB loads B/C with 0027H and then performs a jump to the point in RESTA just past the place in that subroutine where it loads B/C with a similar LXI instruction. In this way, the same subroutine does double duty for two different situations. This is a common method among experienced programmers for reducing program size. You should study the method as illustrated in the program beginning at 0390H.